METHOD AND MECHANISM FOR HANDLING ARBITRARILY-SIZED XML IN SQL OPERATOR TREE

Inventors:

James Warner

Muralidhar Krishnaprasad

Anand Manikutty

Zhen Liu

Ravi Murthy

Susan Kotsovolos

Vikas Arora

Assignee:

Oracle International Corporation
500 Oracle Parkway
Redwood Shores, California 94065

Prepared By:

Peter C. Mei
Bingham McCutchen LLP
Three Embarcadero Center, Suite 1800
San Francisco, California 94111
(650) 849-4870

Express Mail Label No. EV348160129US

METHOD AND MECHANISM FOR HANDLING ARBITRARILY-SIZED XML IN

SQL OPERATOR TREE

BACKGROUND AND SUMMARY

5   **[0001]**   The invention relates to a method and mechanism for efficiently accessing and generating data for a set of operators. In addition, an embodiment of the invention relates to a method and mechanism for efficiently accessing and generating arbitrarily-sized XML in a SQL operator tree.

[0002]   In the computing context, operators act upon one or more inputs to perform

10   arithmetic and/or logic tasks. To illustrate a very simple example, consider the following expression:

$$(a + b) - c$$

This expression contains two operators. The operator '+' performs an addition function. The operator '-' performs a subtraction function.

15   **[0003]**   Requirements may exist regarding the order in which the operators are evaluated in an expression. In some cases, the order of operation is implicit based upon the types of operators present in a statement, the left-right ordering of operators in a statement, and upon the relative/default ordering that is specified by a given system. The ordering of operators is important if there is a possibility of side effects if the operators are evaluated in

20   a different order. In addition, the expression may contain an explicit indication of the order in which operators are evaluated. For example, in the sample expression above, the '(' and ')' symbols explicitly indicate that the content between these symbols (i.e., the 'a+b' operation) should be performed first, with the results subsequently used to evaluate with the '- c' operation. Therefore, for this example expression, the first operator '+' performs an

25   addition function upon the values of 'a' and 'b'. The second operator '-' subtracts the value of 'c' from the result of the '+' operation.

[0004]   Fig. 1 shows an expression tree 100 (also called an operator tree) that is associated with this expression. The highest level of the tree is the node 102 representing the '-' operator. Extending into node 102 from the left side is the output from the child node 104 representing the '+' operator. Extending into node 102 from the other side is the child node 106 representing the 'C' value. Drilling down lower into this expression tree, it can be seen that branching into node 104 from the left side is the child node 108 representing the 'A' value. Branching into the node 104 from the right side is the child node 110 representing the 'B' value.

[0005]   The typical approach for processing this type of expression tree is to use "bottom-up" evaluations of operators. With the bottom-up approach, the lower levels of the expression tree are processed first, with the intermediate results propagated upward as each higher level of the expression tree is subsequently evaluated. The bottom-up approach is used to ensure that the correct order of evaluation is followed. In the example expression tree 100 of Fig. 1, this means that the operator at node 104 is evaluated first based upon inputs from 108 and 110. The intermediate result from the operator at node 104 is propagated upwards to be evaluated by the operator at node 102 along with the input from 106. The output from node 102 is the final result (unless this expression tree 100 is a sub-tree to a larger expression tree, in which case the out put from node 102 becomes an intermediate result that is itself propagated upwards to one or more other higher levels).

[0006]   When processing expression trees using the bottom-up approach, each set of intermediate results from lower levels of the tree may be buffered into temporary storage locations so that they can be accessed and used by operators at higher levels of the tree. A problem with this approach is that the requirement to store the intermediate results into buffers could be relatively expensive, particularly for complex expressions when there are multiple levels of operators that will require multiple levels of buffers.

[0007]   Moreover, if some or all of the intermediate results stay unchanged when moving up the hierarchy of operator levels, then the same set of data may be copied over and over again while the expression tree is being evaluated. Creating these multiple copies

could be expensive, particularly if the data size increases while progressing up the levels of the expression tree.

[0008]   An example of how this problem occur in the real world exists with respect to the desire of many modern computing systems to add XML functionality to be able to create, store, and retrieve data in the form of XML from relational or object-relational databases. XML (the "extensible markup language") is a meta-language developed and standardized by the World Wide Web Consortium (W3C) that permits use and creation of customized markup languages for different types of documents. XML is a variant of and is based on the Standard Generalized Markup Language (SGML), the international standard meta-language for text markup systems that is also the parent meta-language for the Hyper-Text Markup Language (HTML).   Since its adoption as a standard language, XML has become widely used to describe and implement many kinds of document types.

[0009]   SQL/XML is a standard that lays out a set of operators for generating XML from a relational database. For example, SQL/XML defines a standard API call, referred to as XMLElement( ), that acts as an operator to generate a XML statement from a data input. However, when this type of operator is used to generate XML, multiple levels of nesting may occur because of the types of data that is being generated. When using the bottom-up approach to handle an expression tree corresponding to this nesting of XMLElement( ) operators, it is quite possible that the same data will be copied, buffered, and propagated through multiple levels of the expression tree. Therefore, if there are ten levels of nesting in the operator tree, ten copies of the lowest-level data is generated, buffered, and copied, with nine additional copies at the next highest level, and eight additional copies at the next highest level, etc. As is evident, this type of approach could be quite inefficient.

[0010]   The present invention provides an improved   method and mechanism for processing expressions and operator trees. An embodiment of the invention is particularly useful to optimize processing of XML statements with respect to SQL operators. A top-down processing approach can be taken to directly output data from operators to a data stream. In addition, multiple processing approaches can be taken within a single expression tree, with some operators processed using the top-down approach and other

operators processed with the bottom-up approach. Even data that cannot be streamed is copied fewer times using this approach, and intermediate values from bottom-up processing may still be streamed if it is used by an operator that is eligible for top-down processing.

[0011] Further details of aspects, objects, and advantages of the invention are
5   described below in the detailed description, drawings, and claims.

BRIEF DESCRIPTION OF THE DRAWINGS

[0012]   The accompanying drawings are included to provide a further understanding of the invention and, together with the Detailed Description, serve to explain the principles of the invention. The same or similar elements in the figures may be referenced using like reference numbers.

[0013]   Fig. 1 shows an example expression tree.

[0014]   Fig. 2 depicts a flowchart of a process for processing an expression according to an embodiment of the invention.

[0015]   Fig. 3A shows another example expression tree.

[0016]   Fig. 3B illustrates the application of the process of Fig. 2 to the expression tree of Fig. 3A.

[0017]   Fig. 3C illustrates a bottom-up approach to processing the expression tree of Fig. 3A.

[0018]   Fig. 4 depicts a flowchart of a process for processing handling XML in a SQL-based operator tree.

[0019]   Fig. 5 shows an example SQL-based operator tree for XML.

[0020]   Fig. 6A-I illustrates the application of the process of Fig. 4 to the expression tree of Fig. 5.

[0021]   Fig. 7 is a diagram of a computer system with which the present invention can be implemented.

DETAILED DESCRIPTION

[0022]   The present invention provides a method and mechanism for efficiently accessing and generating data for a set of operators. In addition, an embodiment of the invention relates to a method and mechanism for efficiently accessing and generating

5   arbitrarily-sized XML in a SQL operator tree.

[0023]   According to a first embodiment of the invention, the present method creates a stream to which operator results will be placed. This stream could be built on top of, for example, a buffer, a LOB (large object), or another type of stream such as an HTTP stream. Examples of other types of streams that could be built upon include a segmented

10   array, data socket, pipe, files, and networking/internet streams such as an FTP stream. The operators are evaluated in a top-down, depth-first manner, each outputting to this stream. In this manner, wasteful copies of data are avoided and performance is significantly improved.

[0024]   Fig. 2 shows a flowchart of a process for evaluating an expression tree according to an embodiment of the invention. At 202, a determination is made whether an

15   already-existing data stream has been identified to the evaluation function. If not, then a new data stream is created at 204. This type of identification for an already-existing data stream will be passed in, for example, if the evaluation function is being recursively called by another execution of the evaluation function at a different level of the expression tree.

[0025]   At 203, a child of the operator is identified and reviewed. The specific

20   ordering in which the child nodes of the operator are reviewed is subject to the type and content of the particular expression under review, as well as the configuration and requirements of the system to which the invention is directed. Based upon whatever order is required for the expression, a first child node is selected at 203.

[0026]   A determination is made whether the selected child node corresponds to an

25   optimizable operator (208). According to one approach, an example category of such optimizable operators are operators which can be evaluated in a top-down manner. If the selected operator is an optimizable operator, then a new execution of the process (beginning again at 202) is implemented for that child operator. In effect, the operator is called with the

present process and the present data stream is passed in. This can be implemented such that the present process is recursively called for the child operator with the present stream passed to the recursive execution. As is clear to those skilled in the art, the present process is not restricted to recursive processing, since other approaches known in the art can be taken to

5   provide a concurrent execution of the process for a child node.

[0027]   If the selected child node corresponds to an operator that is not optimizable (206), then it is evaluated in a default manner (212). For example, if the child node refers to an arithmetic operator such as '+' or '-', then the child operator is not evaluated using the top-down approach. Such operators may be evaluated, for example, using the bottom-up

10   approach.

[0028]   The output from these operators can be placed in a temporary storage location and used by the parent node to generate an output to the data stream (214). Depending upon the type of output produced or type of operator at the parent node and whether the output is suitable for streaming, the output from the non-optimizable node may

15   sometimes be directly placed in the data stream. If the selected child node is not an operator, then the data at the child is directly placed in the data stream (214).

[0029]   It is noted that when processing an operator that is not optimizable, it is possible that sub-children of that node may be optimizable. According to an embodiment, the optimizable sub-children may themselves call the present process to perform top-down

20   processing for the relevant portions of the expression tree. Therefore, different types of processing may be used at different levels of the operator tree.

[0030]   A determination is made whether any more child nodes exist for the present operation (216). If so, then the process returns back to 203 to process the additional child node. If there are no more child nodes to process, and if the current operator created the

25   data stream, then the data stream is closed.

[0031]   To illustrate this process, consider the following expression:

Concat((Concat(Concat('ABC', 'DEF'), 'HIJ')), (1+2))

For the purposes of this example, assume the function Concat(a, b) performs the action of concatenating the values of a and b together. Further assume that this function is capable of concatenating numerical values along with string values.

[0032]   Fig. 3A shows an example operator tree 300 corresponding to this expression. Node 302 is the top-most node, which corresponds to a Concat() function having a first child node 304 and a second child node 308. Child node 308 corresponds to the '+' operator. Child 304 corresponds to another Concat() function having a first sub-child node 306 and a second child corresponding to the value 'HIJ'. The sub-child node 306 corresponds to the Concat() function with inputs values of 'ABC' and 'DEF'.

[0033]   Fig. 3B illustrates how the method of Fig. 2 can be applied to optimize the processing of this expression. For the sake of explanation, assume that the Concat() function is an optimizable operator that can be processed in a top-down manner, in which each data argument to the function can be output to a stream in its proper order. The process begin at the node 302, where a first determination is made regarding the existence of a data stream. A data stream is created if it does not yet exist.

[0034]   Next, a determination is made whether any child node to node 302 is an optimizable operator. Here, the first child node 304 corresponds to the Concat() function, which is an optimizable operator. Therefore, the process proceeds to node 304 and another execution of the process is recursively called.

[0035]   At the recursive processing for node 304, a determination is made regarding the existence of a data stream. Here, the data stream created with respect to node 302 can be passed to the recursive processing for its child node 304. Therefore, a new data stream does not have to be created. Next, a determination is made whether a first sub-child node exists with respect to node 304 that is itself an optimizable operator. Here, the first sub-child node 306 corresponds to the Concat() function, which is an optimizable operator. Therefore, the process proceeds to node 306 and another execution of the process is recursively called.

[0036]   At the recursive processing for node 306, another determination is made regarding the existence of a data stream. The data stream had already been created with

respect to node 302, which has been passed to the recursive processing for node 304, and will again be passed to the recursive processing for node 306. Thus, a new data stream does not have to be created.

[0037]   Next, a determination is made whether a first sub-child node exists with respect to node 306 that is itself an optimizable operator. There are no child nodes to node 306 that corresponds to an optimizable operator. Therefore, the process performs normal processing of the Concat() function corresponding to node 306 but places the output to the data stream. The input data values for node 306 'ABC' and 'DEF' are output to the data stream, without requiring these values or the concatenated results to be placed in a temporary storage location or buffer.

[0038]   At this point, the data stream includes the following: 'ABCDEF'.

[0039]   Once processing for node 306 has been completed, the processing return back to its parent node 304. A determination is made whether any more child nodes exist for node 304 which are themselves optimizable operators. There are no additional child nodes to node 304 that corresponds to an optimizable operator. The process continues normal processing of the Concat() function corresponding to node 304 that places the input value 'HIJ' from child 312 directly to the data stream, without requiring this value or the concatenated results of the node to be placed in a temporary storage location or buffer.

[0040]   At this point, the data stream has the following: 'ABCDEFHIJ'.

[0041]   Once processing for node 304 has been completed, the processing returns back to its parent node 302. A determination is made whether any more child nodes exist for node 302 which are themselves optimizable operators. There are no additional child nodes to node 302 that corresponds to an optimizable operator.

[0042]   However, the other child node 308 to parent node 302 corresponds to an operator that is not to be optimized in a top-down manner. Therefore, the operator at node 308 is evaluated in another appropriate or default manner, e.g., using the bottom-up approach. The (1+2) expression rooted at node 308 is evaluated to generate the result 3 which is placed in a temporary storage location, e.g., a buffer. The concat() operator at the

parent node 302 takes the result "3" from the buffer and outputs this value to the data stream.

[0043] At the end of this processing, the data stream now has the following: 'ABCDEFHIJ3'.

[0044] Therefore, it can be seen that the present process can be configured to allow both top-down and bottom-up processing for the same expression. If node 308 itself contains sub-child nodes that correspond to optimizable operators, then those operators can also be evaluated using the present optimizing process.

[0045] To highlight and contrast some of the advantages of this process, reference is made to Fig. 3C which illustrates how this same expression could be evaluated using only the bottom-up approach. In this type of approach, the lower levels of the tree 300 are evaluated first, with intermediate results parked in temporary storage locations that are available to be accessed by operators at higher levels of the tree 300.

[0046] For the present expression, the lowest level of the tree 300 involves evaluation of node 306 corresponding to a Concat() function to concatenate the 'ABC' and 'DEF' values. The result of this evaluation is the 'ABCDEF' value which becomes an input to node 304 at a higher level of the expression tree 300. This value 'ABCDEF' is buffered at a temporary storage location 310.

[0047] The next higher level of the expression tree 300 at node 304 is now evaluated. Node 304 corresponds to a Concat() function that concatenates the output from node 306 with the value 'HIJ'. For this type of bottom-up processing, this means that the a concatenation operation is performed against the stored output value 'ABCDEF' from node 306 at storage location 310 with the data value 'HIJ' to produce the output value 'ABCDEFHIJ'. This intermediate output value will become an input to an operator at the parent node 302 at a higher level of the expression tree. This value 'ABCDEFHIJ' is buffered at a temporary storage location 312.

[0048] At this same level of the expression tree 300, the operator at node 308 is now evaluated. Node 308 corresponds to a '+' operation that adds the input values 1 and 2. This produces an intermediate output value of 3 which will become an input to an operator

at the parent node 302 at a higher level of the expression tree. The intermediate value 3 is buffered at a temporary storage location 314.

[0049]   The next higher level of the expression tree 300 at node 302 is now evaluated. Node 302 corresponds to a Concat() function that concatenates the output from node 304 with the result of the operator at node 308. This means that a concatenation operation is performed against the stored output value 'ABCDEFHIJ' from node 304 at storage location 312 with the stored value 3 from node 308 at storage location 314. Therefore, the Concat() operator at node 302 will produce an output value of 'ABCDEFHIJ3' which may itself be placed in a temporary storage location 316.

[0050]   Fig. 3C highlights several significant disadvantages with the bottom-up approach. First, the bottom-up approach will require intermediate results to be generated and stored into temporary storage locations, such as storage locations 310 and 312. For a complex expression, a significant amount of system resources may be required to perform the actions of copying to and reading from these storage locations, in addition to the expense related to allocating these storage locations. Moreover, it is noted that the exact same data values are copied numerous times throughout this process. For example, the 'ABC' and 'DEF' values are repeated at storage locations 310, 312, and 316. The 'HIJ' data value is repeated at storage locations 312 and 316. Repeating the same data values in different storage locations in this manner is highly inefficient and could consume an excessive amount of system resources. This is particularly true for complex expressions having numerous levels of operators, where values may be repeated, for example, on a linear basis O(d) where d is the depth of the operator tree.

[0051]   In contrast, the approach of the present embodiment of the invention is performed with zero or minimum buffering of intermediate results, since intermediate data is directly placed in the output data stream. Even data that cannot or should not be streamed is copied fewer times using this process. The data stream can be configured to be any location or structure, such as an HTTP stream, buffer, large object (LOB), or even the intended persistent location for the data (so that a data copy to that location does not have to be performed). Examples of other types of streams that could be built upon include a

segmented array, data socket, pipe, files, and networking/internet streams such as an FTP stream.

[0052] The present invention is particularly useful for handling arbitrarily sized XML in a SQL operator tree. This type of processing may occur, for example, when making SQL/XML calls to generate XML from a relational database. As noted above, SQL/XML is a standard that lays out a set of operators for generating XML from a relational database.

[0053] An example SQL/XML call that performs this type of processing is the XMLElement( ) function. Based upon an input value, the XMLElement function generates an XML element. For example, the following SQL statement using this function:

```
SELECT  XMLElement( "Element", column_value )
FROM  Example_Table;
```

that access the following Example Table:

Column_Value
2
3

will read each row from this example table, apply the XMLElement function to that row to create an XML element, and output each generated XML element as follows:

```
<Element>  2  </Element>
<Element>  3  </Element>
```

[0054] In typical usage, when this type of operator is used to generate XML, multiple levels of nesting may occur because of the complex types, schemas, and formats of data that is normally used by modern organizations and businesses. When using the bottom-up approach to handle an expression tree corresponding to this nesting of XMLElement( ) operators, it is quite possible that the same data will be copied, buffered, and propagated

through multiple levels of the expression tree. As is evident, this type of bottom-up approach could be quite inefficient.

[0055]   Fig. 4 shows a flowchart of a process that can be used to generate XML from a SQL operator tree, e.g., for a statement or expression that uses this or a similar type

5   of SQL/XML operator. At 402, a determination is made whether an already-existing data stream has been identified to the evaluation function. If not, then a new data stream is created at 404. This stream can be created over any structure or location, depending upon the context in which the invention is used, including for example, a buffer, LOB, segmented array, data socket, pipe, files, and networking/internet streams such as anFTP stream or

10   HTTP stream.

[0056]   A determination can be made whether an opening XML tag needs to be generated. If so, then an opening XML tag is generated at 405. For example, the XMLElement() functions generates this tag to satisfy the syntactic requirement of XML for a representation of an element. If the operator is XMLElement("A", op) where op is some

15   further operator tree, then "<A>" is printed.

[0057]   At 403, a child of the XML-based operator is identified and reviewed. A determination is made whether the selected child node corresponds to an optimizable operator (408). For purposes of the present embodiment, optimizable operators are those operators which can be evaluated in a top-down manner. An example of such an

20   optimizable operator includes SQL/XML operators such as XMLElement(), XMLAgg(), XMLConcat(), XMLForest(), XMLAttribute(), XMLComment(), XMLPI(), and other operators that may be created for this SQL/XML standard or other standards in the future. Other examples of optimizable operators include, for example, a concatenation operator or a CASE WHEN statement on top of a SQL/XML operator. In addition, other operators that

25   return values linearly, e.g., operators that linearly return XML values to a buffer, could also qualify as optimizable operators. If the selected operator is an optimizable operator, then a new execution of the process (beginning again at 402) is implemented for the child operator. This can be implemented by recursively calling the evaluation function while passing an identification of the present data stream (410).

[0058] If the selected child node corresponds to an operator that is not optimizable (406), then it is evaluated in a default manner (412). According to an embodiment, this includes column data and non-optimizable operators. These child nodes are evaluated in the normal way, and the data is copied in. The output from these operators can be placed in a temporary storage location and used by the parent node to generate an output to the data stream (414). Depending upon the type of output produced or type of operator at the parent node and whether the output is suitable for streaming, the output from the non-optimizable node may sometimes be directly placed in the data stream. If the selected child node is not an operator, then the data at the child is directly placed in the data stream (414).

[0059] A determination is made whether any more child nodes exist for the present operation (416). If so, then the process returns back to 403 to process the additional child node.

[0060] Once all child nodes have been processed, a determination is made whether a closing XML tag is needed. If so, then the appropriate closing XML tag is generated at 418. For example, if the operator is XMLElement("A", op) where op is some further operator tree, then "</A>" is printed.

[0061] At this point, if the current operator created the data stream, then the data stream is closed.

[0062] Consider if it is desired to generate an XML document relating to purchase orders used by a fictional company, with the XML data having the following structure:

```
<PurchaseOrder>
        <OrderNo> Value_of_Order_Number </OrderNo>
        <Address>
                <State> Value_of_State </State>
                <City> Value_of_City </City>
        </Address>
</PurchaseOrder>
```

[0063] Assume that the following SQL statement is used to generate this type of XML:

```
SELECT XMLElement ( "PurchaseOrder",
```

```
        XMLElement ( "OrderNo", OrderNoVal),
        XMLElement ( "Address",
                XMLElement ( "State", StateVal),
                XMLElement ( "City", CityVal)
                )
        )
    FROM Example_Table_2;
```

An example table that is accessed by this statement may have the following column structure:

OrderNoVal	StateVal	CityVal

[0064] Fig. 5 shows an example operator tree 500 that corresponds to this SQL statement. In operator tree 500, node 504 corresponds to the main XMLElement() operator for the "PurchaseOrder" element. Node 506 corresponds to the XMLElement() operator for the "OrderNo" element. Node 508 corresponds to the XMLElement() operator for the "Address" element, with its child-nodes 510 and 512 corresponding to the XMLElement() operator for the "State" and "City" elements, respectively.

[0065] Figs. 6A-I illustrate how the process of Fig. 4 can be used to evaluate operator tree 500. Referring to Fig. 6A, the process begins by starting the evaluation of the top-level node 504. An initial determination is made regarding the existence of a data stream. Assume that a data stream does not yet exist, and therefore data stream 602 is created.

[0066] Because node 504 relates to the XMLElement() operator, an opening XML tag is created and placed in the data stream. For this example, the opening XML tag is the following: "<PurchaseOrder>".

[0067] A determination is made whether node 504 corresponds to any child nodes. Here a node 506 is a first child node to parent node 504. A determination is made whether node 506 corresponds to an optimizable operator. In this example, node 506 corresponds to

the XMLElement() operator, which is an optimizable operator. Therefore, the operator corresponding to node 506 is called and the present data stream 602 is passed to the processing of node 506.

[0068] Fig. 6B illustrates the initial processing of node 506, which corresponds to the XMLElement() operator for the "OrderNo" element. An opening XML tag is generated and placed in the data stream 602 that corresponds to this element, as follows: "<OrderNo>". Next, a determination is made whether any child node to node 506 is an optimizable operator. Here, the only child to node 506 is a value (i.e., OrderNoVal) from a database column. The data value for the OrderNoVal column is retrieved and placed in the data stream 602. Since there are no further children for node 506, a closing XML tag is generated as follows: "</OrderNo>".

[0069] Referring to Fig. 6C, the process returns back to parent node 504. A determination is made whether node 504 is associated with any more child nodes. In this example, node 508 is another child node to parent node 504. A determination is made whether node 508 corresponds to an optimizable operator. In this example, node 508 corresponds to the XMLElement() operator, which is an optimizable operator. The operator corresponding to node 508 is called and the present data stream 602 is passed to the processing of node 508.

[0070] Turning to Fig. 6D, processing for node 508 will begin. Because node 508 relates to the XMLElement() operator for the "Address" element, an opening XML tag is created and placed in the data stream. For this example, the opening XML tag is the following: "<Address>".

[0071] A determination is made whether node 508 corresponds to any child nodes. A node 510 is a first child node to node 508. A determination is made whether node 510 corresponds to an optimizable operator. In this example, node 510 corresponds to the XMLElement() operator, which is an optimizable operator. Therefore, the operator corresponding to node 510 is called and the present data stream 602 is passed to the processing of node 510.

[0072]   Fig. 6E illustrates the processing of node 510, which corresponds to the XMLElement() operator for the "State" element. An opening XML tag is generated and placed in the data stream 602 that corresponds to this element, as follows: "<State>". Next, a determination is made whether any child node to node 510 is an optimizable operator.

5   Here, the only child to node 510 is a value (i.e., StateVal) from a database column. The data value for the StateVal column is retrieved and placed in the data stream 602. Since there are no further children for node 510, a closing XML tag is generated as follows: "</State>".

[0073]   Referring to Fig. 6F, the process returns back to node 508. A determination is made whether node 508 is associated with any more child nodes. In this example, node

10   508 is another child node 512. In this example, node 512 corresponds to the XMLElement() operator, which is an optimizable operator. Therefore, the operator corresponding to node 512 is called and the present data stream 602 is passed to the processing of node 512.

[0074]   Fig. 6G illustrates the processing of node 512, which corresponds to the XMLElement() operator for the "City" element. An opening XML tag is generated and

15   placed in the data stream 602 that corresponds to this element, as follows: "<City>". Next, a determination is made whether any child node to node 512 is an optimizable operator. The only child to node 512 is a value (i.e., CityVal) from a database column. Thus, the data value for the CityVal column is retrieved and placed in the data stream 602. Since there are no further children for node 512, a closing XML tag is generated as follows: "</City>".

20   [0075]   Referring to Fig. 6H, the process returns back to the process of node 508. A determination is made whether node 508 is associated with any more child nodes. In this example, there are no further child nodes associated with node 508. A closing XML tag is generated for node 508 as follows: "</Address>". Processing now ends for node 508, and the process returns back to its parent node 504.

25   [0076]   Fig. 6I shows the final processing that occurs for node 504. A determination is made whether node 504 is associated with any more child nodes. In this example, there are no further child nodes associated with node 504. A closing XML tag is generated for node 508 as follows: "</PurchaseOrder>". Processing now ends for node 508, and the data stream 602 can be closed.

[0077]   Therefore, what has been disclosed is an improved method and mechanism for processing expressions and operator trees. An embodiment of the invention is particularly useful to optimize processing of XML statements with respect to SQL operators. A top-down processing approach can be taken to directly output data from operators to a

5   data stream. In addition, multiple processing approaches can be taken within a single expression tree, with some operators processed using the top-down approach and other operators processed with the bottom-up approach. Even data that can not be streamed is copied fewer times using this approach, intermediate values from bottom-up processing may still be streamed if it is used by an operator that is eligible for top-down processing.

10

SYSTEM ARCHITECTURE OVERVIEW

[0078]   The execution of the sequences of instructions required to practice the invention may be performed in embodiments of the invention by a computer system 1400 as shown in Fig. 7. As used herein, the term computer system 1400 is broadly used to describe

15   any computing device that can store and independently run one or more programs. In an embodiment of the invention, execution of the sequences of instructions required to practice the invention is performed by a single computer system 1400. According to other embodiments of the invention, two or more computer systems 1400 coupled by a communication link 1415 may perform the sequence of instructions required to practice the

20   invention in coordination with one another. In order to avoid needlessly obscuring the invention, a description of only one computer system 1400 will be presented below; however, it should be understood that any number of computer systems 1400 may be employed to practice the invention.

[0079]   Each computer system 1400 may include a communication interface 1414

25   coupled to the bus 1406. The communication interface 1414 provides two-way communication between computer systems 1400. The communication interface 1414 of a respective computer system 1400 transmits and receives signals, e.g., electrical, electromagnetic or optical signals, that include data streams representing various types of

information, e.g., instructions, messages and data. A communication link 1415 links one

computer system 1400 with another computer system 1400. A computer system 1400 may

transmit and receive messages, data, and instructions, including program, i.e., application,

code, through its respective communication link 1415 and communication interface 1414.

5   Received program code may be executed by the respective processor(s) 1407 as it is

received, and/or stored in the storage device 1410, or other associated non-volatile media,

for later execution.

[0080]   In an embodiment, the computer system 1400 operates in conjunction with

a data storage system 1431, e.g., a data storage system 1431 that contains a database 1432

10   that is readily accessible by the computer system 1400. The computer system 1400

communicates with the data storage system 1431 through a data interface 1433. A data

interface 1433, which is coupled to the bus 1406, transmits and receives signals, e.g.,

electrical, electromagnetic or optical signals, that include data streams representing various

types of signal information, e.g., instructions, messages and data. In embodiments of the

15   invention, the functions of the data interface 1433 may be performed by the communication

interface 1414.

[0081]   Computer system 1400 includes a bus 1406 or other communication

mechanism for communicating instructions, messages and data, collectively, information,

and one or more processors 1407 coupled with the bus 1406 for processing information.

20   Computer system 1400 also includes a main memory 1408, such as a random access

memory (RAM) or other dynamic storage device, coupled to the bus 1406 for storing

dynamic data and instructions to be executed by the processor(s) 1407. The main memory

1408 also may be used for storing temporary data, i.e., variables, or other intermediate

information during execution of instructions by the processor(s) 1407. The computer system

25   1400 may further include a read only memory (ROM) 1409 or other static storage device

coupled to the bus 1406 for storing static data and instructions for the processor(s) 1407. A

storage device 1410, such as a magnetic disk or optical disk, may also be provided and

coupled to the bus 1406 for storing data and instructions for the processor(s) 1407. A

computer system 1400 may be coupled via the bus 1406 to a display device 1411, such as,

but not limited to, a cathode ray tube (CRT), for displaying information to a user. An input device 1412, e.g., alphanumeric and other keys, is coupled to the bus 1406 for communicating information and command selections to the processor(s) 1407.

[0082]   According to one embodiment of the invention, an individual computer
5   system 1400 performs specific operations by their respective processor(s) 1407 executing one or more sequences of one or more instructions contained in the main memory 1408. Such instructions may be read into the main memory 1408 from another computer-usable medium, such as the ROM 1409 or the storage device 1410. Execution of the sequences of instructions contained in the main memory 1408 causes the processor(s) 1407 to perform the
10   processes described herein. In alternative embodiments, hard-wired circuitry may be used in place of or in combination with software instructions to implement the invention. Thus, embodiments of the invention are not limited to any specific combination of hardware circuitry and/or software.

[0083]   The term "computer-usable medium" or "computer-readable medium" as
15   used herein, refers to any medium that provides information or is usable by the processor(s) 1407. Such a medium may take many forms, including, but not limited to, non-volatile, volatile and transmission media. Non-volatile media, i.e., media that can retain information in the absence of power, includes the ROM 1409, CD ROM, magnetic tape, and magnetic discs. Volatile media, i.e., media that can not retain information in the absence of power,
20   includes the main memory 1408. Transmission media includes coaxial cables, copper wire and fiber optics, including the wires that comprise the bus 1406. Transmission media can also take the form of carrier waves; i.e., electromagnetic waves that can be modulated, as in frequency, amplitude or phase, to transmit information signals. Additionally, transmission media can take the form of acoustic or light waves, such as those generated during radio
25   wave and infrared data communications.

[0084]   In the foregoing specification, the invention has been described with reference to specific embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention. For example, the reader is to understand that the specific

Patent
OI7035752001

ordering and combination of process actions shown in the process flow diagrams described herein is merely illustrative, and the invention can be performed using different or additional process actions, or a different combination or ordering of process actions. The specification and drawings are, accordingly, to be regarded in an illustrative rather than restrictive sense.